TP12 – Réalité diminuée

On appelle *réalité diminuée* le processus inverse de la *réalité augmentée*. Cette tâche, qui consiste à supprimer un ou plusieurs objets d'une scène, est parfois utilisée pour de mauvaises raisons, par exemple pour éliminer une personne indésirable sur une photographie, mais elle sert également à retoucher une image, par exemple pour « gommer » le mobilier dans un appartement afin de faciliter sa visite virtuelle.

Limites de l'*inpainting* par diffusion

La méthode d'*inpainting* vue dans le TP11 s'appelle l'*inpainting par diffusion*. Elle permet de restaurer des zones de faible épaisseur, comme les rayures sur une photographie ancienne, mais ne permet pas de répondre aux besoins de la réalité diminuée. Faites une copie du script `exercice_2` du TP11, de nom `exercice_0`, et apportez les modifications suivantes : fixez le paramètre `lambda` à 1 ; remplacez les fichiers `fleur_avec_defaut.png` et `masque_fleur.png` par `randonneur.jpg` et `masque_randonneur.png`, respectivement. Vous constatez que le résultat n'est pas satisfaisant. La réalité diminuée nécessite d'utiliser des méthodes d'*inpainting* qui, contrairement à l'*inpainting* par diffusion, ne découlent pas d'une approche variationnelle.

Principe de l'*inpainting* par rapiéçage

Si Ω désigne l'ensemble des pixels de l'image, et D le domaine à compléter, l'*inpainting* « par rapiéçage » (*patch-based inpainting*) consiste à rechercher, en tout pixel $\mathbf{p}$ de la frontière ∂D de D, le pixel $\widehat{\mathbf{q}} \in \overline{D} = \Omega \backslash D$ (complémentaire de D à Ω) dont le voisinage $V(\widehat{\mathbf{q}})$ « ressemble le plus » au voisinage $V(\mathbf{p})$ de $\mathbf{p}$. En pratique, le voisinage est une fenêtre centrée de taille $(2\,t+1) \times (2\,t+1)$, $t > 0$. Si $\mathbf{p} = (i_\mathbf{p}, j_\mathbf{p})$ est un pixel de ∂D, notons $R(\mathbf{p})$ l'ensemble des *indices relatifs* (i, j) des pixels voisins $\mathbf{p}' = (i_\mathbf{p} + i, j_\mathbf{p} + j)$ se trouvant hors de D :

$$R(\mathbf{p}) = \{(i,j) \in \{-t, \ldots, t\}^2 \,/\, (i_\mathbf{p} + i, j_\mathbf{p} + j) \in \overline{D}\} \tag{1}$$

Comme au moins un voisin de $\mathbf{p} \in \partial D$ appartient à $\overline{D}$, il s'ensuit que $\mathrm{Card}(R(\mathbf{p})) > 0$. Pour une image en niveaux de gris I, la *dissemblance* $d(\mathbf{p}, \mathbf{q})$ entre le voisinage d'un pixel $\mathbf{p} \in \partial D$ et le voisinage d'un pixel $\mathbf{q} \in \overline{D}$ est définie comme suit (cette définition est facile à étendre aux images en couleur) :

$$d(\mathbf{p}, \mathbf{q}) = \frac{1}{\mathrm{Card}(R(\mathbf{p}))} \sum_{(i,j) \in R(\mathbf{p})} [I(i_\mathbf{p} + i, j_\mathbf{p} + j) - I((i_\mathbf{q} + i, j_\mathbf{q} + j)]^2 \tag{2}$$

Pour trouver $\widehat{\mathbf{q}}$, il est inutile de calculer $d(\mathbf{p}, \mathbf{q})$ en chaque pixel $\mathbf{q} \in \overline{D}$, car une image est généralement constituée de régions de texture homogène. Nous nous contentons de rechercher $\widehat{\mathbf{q}}$ dans une fenêtre $F(\mathbf{p})$ centrée en $\mathbf{p}$, de taille $(2\,T+1) \times (2\,T+1)$, avec $T > t$. Il suffit même de mener cette recherche pour les pixels $\mathbf{q} \in F'(\mathbf{p})$ tels que $V(\mathbf{q}) \subset \big(F(\mathbf{p}) \cap \overline{D}\big)$, comme le montre la figure 1. Le pixel $\widehat{\mathbf{q}}$ associé à un pixel $\mathbf{p} \in \partial D$ est donc tel que :

$$\widehat{\mathbf{q}} = \underset{\mathbf{q} \in F'(\mathbf{p})}{\mathrm{argmin}} \{d(\mathbf{p}, \mathbf{q})\} \tag{3}$$

Le rapiéçage consiste à remplacer les pixels manquants de $V(\mathbf{p})$ par les pixels de mêmes positions dans $V(\widehat{\mathbf{q}})$.

La première méthode d'inpainting par rapiéçage a été décrite en 2003 par Criminisi, Pérez et Toyama dans un article intitulé *Region filling and object removal by exemplar-based image inpainting*. Bien sûr, de nombreuses améliorations ont été proposées depuis. Une version commerciale de l'inpainting par rapiéçage est proposée dans le logiciel *PaintShop Pro*. Par ailleurs, le greffon (*plugin*) de GIMP de nom *PatchMatch*, disponible dans la boîte à outils G'MIC développée à l'Université de Caen, constitue une version libre de cette méthode d'*inpainting*.

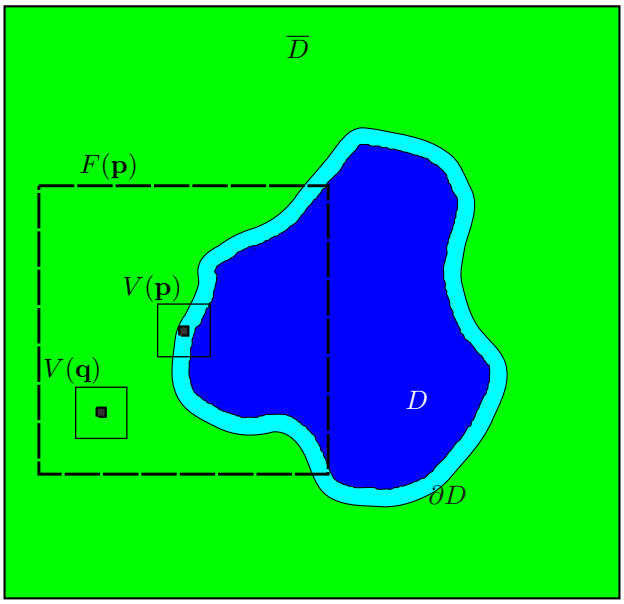


FIGURE 1 – Principe de l'*inpainting* par rapiéçage.

Exercice 1 : traitement des pixels de ∂D par tirage aléatoire

Une première façon de coder la méthode d'*inpainting* par rapiéçage consiste à répéter la boucle suivante, tant que le domaine D n'est pas vide :

1. Choisir un pixel $\mathbf{p}$ de la frontière ∂D par tirage aléatoire.
2. Rechercher le pixel $\widehat{\mathbf{q}}$ défini en (3) en testant tous les pixels $\mathbf{q} \in F'(\mathbf{p})$. Pour chaque $\mathbf{q}$:
   - Calculer la dissemblance $d(\mathbf{p}, \mathbf{q})$ définie par (2).
   - Si $d(\mathbf{p}, \mathbf{q}) < \widehat{d}$, alors $\widehat{\mathbf{q}} \leftarrow \mathbf{q}$ et $\widehat{d} \leftarrow d(\mathbf{p}, \mathbf{q})$.
3. Utiliser $V(\widehat{\mathbf{q}})$ pour compléter les pixels manquants de $V(\mathbf{p})$ par rapiéçage.
4. Mettre à jour D et ∂D.

Écrivez les fonctions `d_min` et `rapiecage_1` permettant au script `exercice_1` de reproduire cet algorithme. En le testant sur l'image `randonneur.jpg`, vous constatez que `exercice_1` fournit des résultats beaucoup plus réalistes que `exercice_0`. Le script `exercice_1_bis` est identique à `exercice_1`, à ceci près que le domaine D à compléter doit être sélectionné par l'utilisateur sous la forme d'un polygone déterminé par une série de clics (double-clic pour terminer). Testez d'abord ce script sur l'image `randonneur.jpg`, puis tentez d'effacer le bateau rouge de l'image `regate.jpg`. Le résultat est décevant, car le remplissage est effectué sans tenir compte de la forme de D, ni de la structure de l'image au voisinage immédiat de D.

Remarque – Lancez le script `explications_frontiere`, dont le but est d'expliquer sur un exemple le fonctionnement de la fonction `frontiere`. Utilisez cette fonction pour calculer ∂D.

Exercice 2 : traitement des pixels de ∂D selon un ordre de priorité

Pour améliorer les résultats du script `exercice_1`, l'idée de Criminisi, Pérez et Toyama consiste à déterminer le prochain pixel $\mathbf{p} \in \partial D$ à traiter selon un ordre de priorité.

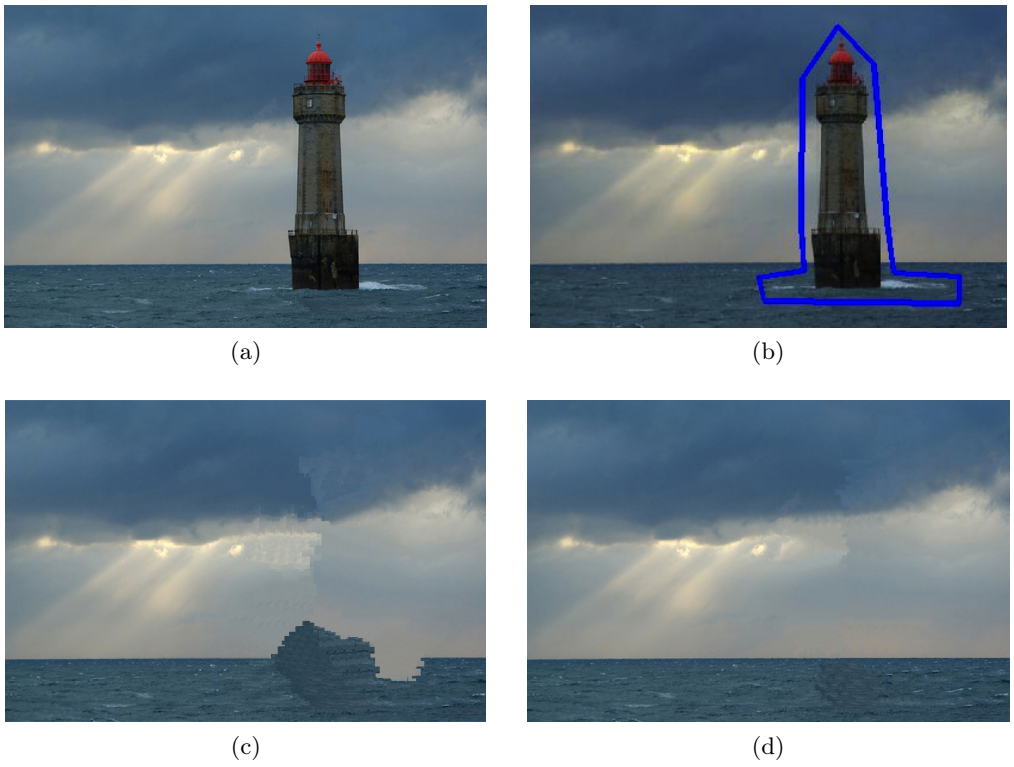
D'une part, il semble pertinent de traiter en premier les pixels situés sur une partie convexe de ∂D, car le nombre de voisins à compléter y est moindre. D'autre part, pour éviter le défaut du résultat de la figure 2-c, il convient de donner priorité aux pixels de ∂D où le gradient de l'image est à la fois élevé et tangent au contour.

La priorité $P(\mathbf{p})$ d'un pixel $\mathbf{p} \in \partial D$ est donc égale au produit de deux coefficients $C(\mathbf{p})$ et $A(\mathbf{p})$:

- À l'initialisation, la *confiance* vaut $C(\mathbf{p}) = 1$ si $\mathbf{p} \in \overline{D}$ (pixel déjà rempli, donc considéré comme fiable), et $C(\mathbf{p}) = 0$ sinon. À chaque rapiéçage, les pixels de $V(\mathbf{p})$ ayant une confiance nulle reçoivent comme nouvelle valeur de la confiance $C(\mathbf{p})$ la confiance moyenne calculée sur $V(\mathbf{p})$.
- L'*attache aux données* $A(\mathbf{p})$ est égale à $|\mathbf{t}(\mathbf{p}) \cdot \nabla u(\mathbf{p})|$, où $\mathbf{t}(\mathbf{p})$ désigne un vecteur *de norme unitaire*, localement tangent au contour de D, et $\nabla u(\mathbf{p})$ le gradient en $\mathbf{p}$ de l'image u en cours de complétion.

Complétez la fonction `priorites`, appelée par le script `exercice_2`, qui doit calculer les priorités et mettre à jour la confiance. La fonction `gradient` de Matlab peut être utilisée pour calculer les vecteurs $\mathbf{t}$ et ∇u. Il est conseillé d'utiliser le canal L (« luminance ») de l'image convertie au format *CIE LAB* (fonction `rgb2lab`) pour calculer ∇u. Il est également conseillé d'utiliser la fonction `dsearchn`, qui est très efficace pour rechercher le point le plus proche d'un point donné. Enfin, n'oubliez pas de convertir l'argument d'entrée `D` au format `double`. Faites ensuite une copie de la fonction `rapiecage_1`, de nom `rapiecage_2`, que vous modifierez de manière à ce que `exercice_2` implémente l'*inpainting* par rapiéçage avec l'ordre de priorité décrit ci-dessus.

Remarque – Une différence notable entre `exercice_1_bis` et `exercice_2` est que, lorsque l'ensemble $F'(\mathbf{p})$ est vide, c'est-à-dire si aucun pixel $\mathbf{q}$ n'est tel que $V(\mathbf{q}) \subset \big(F(\mathbf{p}) \cap \overline{D}\big)$, il ne suffit pas de sauter les étapes 3 et 4, comme le fait `exercice_1_bis`, car cela provoquerait une boucle infinie. Le prochain pixel à traiter dans un tel cas est celui de plus forte priorité pour lequel $F'(\mathbf{p})$ n'est pas vide.



(a) (b) (c) (d)

FIGURE 2 – (a) Image originale. (b) Sélection du domaine D. (c) Résultat sans ordre de priorité (exercice 1), qui comporte un défaut flagrant. (d) Résultat avec ordre de priorité (exercice 2).

Le script `exercice_2` peut procurer de bons résultats, comme cela est illustré sur l'exemple de la figure 2-d. Testez l'image `phare.jpg`, puis d'autres images de votre choix. Notez pour finir que cette nouvelle version de l'*inpainting* par rapiéçage ne suffit pas toujours à répondre aux besoins de la réalité diminuée. Pour vous en convaincre, tentez d'effacer le fauteuil de l'image `mur.jpg`.